Tutoriat 6 (Operating Systems)
CPU Scheduling

Contents

1 Introduction

To maximize CPU utilization, there must always be a process running at all times. When a running process terminates, finishes its time slice, or gets interrupted, the role of the **CPU scheduler** is to immediately allocate the CPU to another *ready* process.

A **CPU scheduler** is a **software component** within the kernel. The scheduler picks the next process to run from the *ready queue* (based on an algorithm), allocates the CPU to the selected process, and handles transitions between states (such as when a process is interrupted by an I/O request).

Recall that a **nonpreemptive** scheduling scheme means that when the CPU is allocated to a process, it runs until it finishes or moves to a waiting state (the OS cannot forcibly take away its CPU time). As for **preemptive** scheduling, the OS can interrupt a running process to allocate the CPU to another process (e.g. when its time slice expires, or there is a process of a higher priority). CPU scheduling decisions take place when a process:

1. Switches from *running state* to *waiting state*.
2. Switches from *running state* to *ready state*.
3. Switches from *waiting state* to *ready state*.
4. Terminates.

For conditions 1 and 4, there is no choice: simply select another process to run. As for conditions 2 and 3, there is a choice: either continue running the current process or select a different one. When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is considered to be **nonpreemptive**; otherwise, it is **preemptive**. However, there is one issue with preemptive scheduling: consider the case of two processes that read and write to the same data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data which are in an inconsistent state. This is known as a **race condition** (and will be studied next time).

A **CPU-I/O burst cycle** is the alternating pattern of a process that executes on the CPU (**CPU burst**) and then waits for an I/O operation to complete (**I/O burst**). The distribution of the CPU bursts is of primary concern (and depends on the scheduling algorithm); as for the I/O bursts, the process cannot directly control those, so they are not taken into account.

The **dispatcher** is a software component within the kernel that gives control of the CPU to the process selected by the CPU scheduler. It is responsible for performing the context switch, switching the CPU from kernel mode to user mode (to execute the user process), and jumping to the proper location in the new process' code to resume its execution from where it last left off. The time it takes to execute these operations is known as the **dispatch latency** and must be minimized.

There are various algorithms for CPU scheduling. These algorithms usually take into account the following criteria:

- **CPU utilization:** the CPU is supposed to be kept busy, and its degree of usage can vary from 0% to 100%. It should be in the range 40% (lightly loaded system) - 90% (heavily loaded system). On Linux/macOS/UNIX systems, CPU utilization can be inspected with the **top** command.
- **Throughput:** the number of processes that complete their execution per time unit.
- **Turnaround time:** the amount of time to execute a particular process.
- **Waiting time:** the amount of time that a process spends waiting in the ready queue.
- **Response time:** the amount of time that a process waits between its arrival and its first chance to execute.

Ultimately, the goal would be to **maximize** CPU utilization and throughput and to **minimize** turnaround time, waiting time, and response time.

2 Scheduling algorithms

2.1 First Come First Served (FCFS)

Processes are executed in the order that they arrive. The algorithm is **nonpreemptive**.

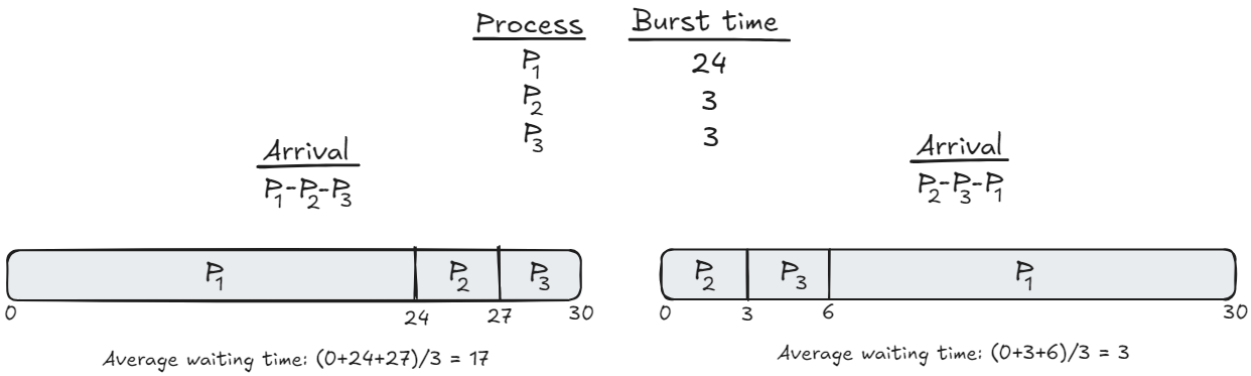

Figure 1: FCFS scheduling

2.2 Priority Scheduling

It is a more general algorithm, where each process is assigned a **priority** value (an integer). A lower value means a higher priority. The CPU scheduler always selects the process with the highest priority to execute; if two or more processes have the same priority, they are typically scheduled using another method (like **FCFS**).

The primary disadvantage is the risk of **starvation:** certain processes with low priorities can perpetually be denied CPU access. If a process has a certain priority and there are always processes in the ready queue with lower priorities, then that process will never have CPU access. A common solution for this is **aging:** a process' priority gradually improves as it sits in the ready queue. Eventually, it will be of high enough priority to execute.

The algorithm can either be **preemptive** (when a process with a higher priority arrives in the ready queue, that process is context switched with the currently executing process) or **nonpreemptive** (the current process keeps executing, even if a higher priority one arrives). The next two algorithms, known as **Shortest Job First Scheduling (SJF, nonpreemptive)** and **Shortest Remaining Time First Scheduling (SRTF, preemptive)**, are two cases of priority scheduling.

2.3 Shortest Job First (SJF)

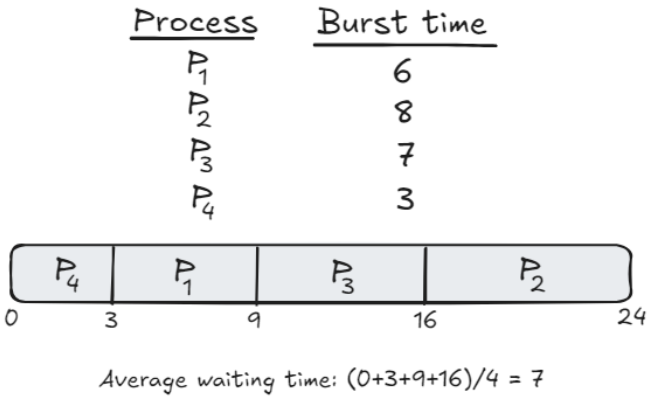

Figure 2: SJF scheduling

The process with the shortest burst time is always picked to execute on the CPU. The burst time can be seen as the **priority** of the process, which is higher when the burst time is lower; the process with the highest priority is executed first. The algorithm is **nonpreemptive**.

Although optimal, the burst time of a process cannot be known before the CPU begins to execute the process. The solution would be to estimate the CPU burst times of each specific process (this method can introduce potential inaccuracies) by using the **most recent actual burst time** and the **previous predicted burst time**. The formula is as follows:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

For a specific process, t_n is the actual length of the n-th CPU burst, τ_{n+1} is the predicted value for the next CPU burst, τ_n is the previous predicted value for the n-th CPU burst, and $0 \leq \alpha \leq 1$ is the weight which allows us to choose whether we want to put emphasis on only the actual CPU burst or on the whole history of previous bursts; the value of α is commonly set to 0.5 (equal emphasis).

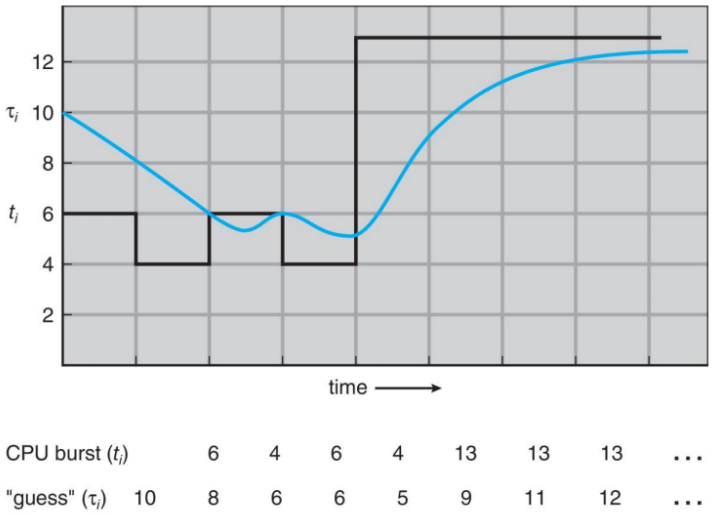

Figure 3: CPU burst prediction

2.4 Shortest Remaining Time First (SRTF)

This is the **preemptive** version of SJF. Whenever a new process arrives in the *ready queue*, the decision on which process to schedule next is redone using SJF. There can also still be the issue of **starvation**.

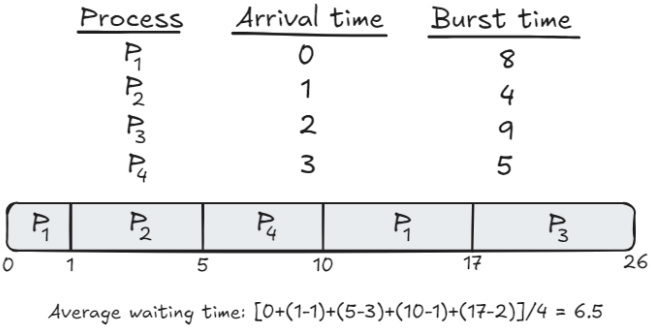

Figure 4: SRTF scheduling

2.5 Round-Robin (RR)

Each process is assigned a fixed amount of time to execute on the CPU, known as a **time quantum** or a **time slice**. Processes are managed in a *circular queue*: when a process' slice expires, it is moved to the end of the queue, and the next process (from the front) gets the CPU. Thus, the algorithm is **preemptive**.

If there are n processes in the ready queue and the time slice is q, then no process will wait more than $q(n-1)$ time units. If q is too large, then the algorithm becomes **FCFS**; if q is too small, then system performance will be degraded due to a large number of context switches.

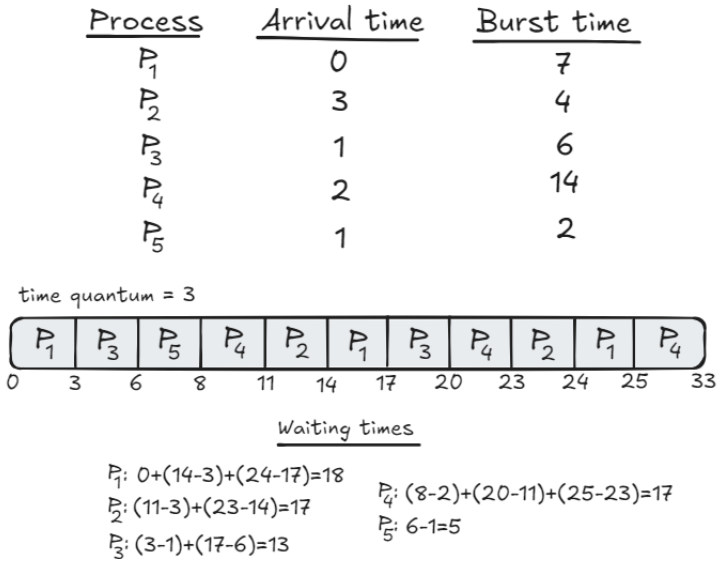

Figure 5: RR scheduling

2.6 Multilevel queue scheduling

When processes can be categorized, the **ready queue** is partitioned into multiple separate queues, each implementing whatever scheduling algorithm is most appropriate. Scheduling must take place not only between the processes of one particular queue, but also between the queues themselves (interqueue scheduling); two common options are **strict priority** (no process in a lower priority queue runs until all higher priority queues are empty) and **round-robin** (each queue gets a time slice). Note that once a process is assigned to a queue, it remains there permanently until it finishes executing.

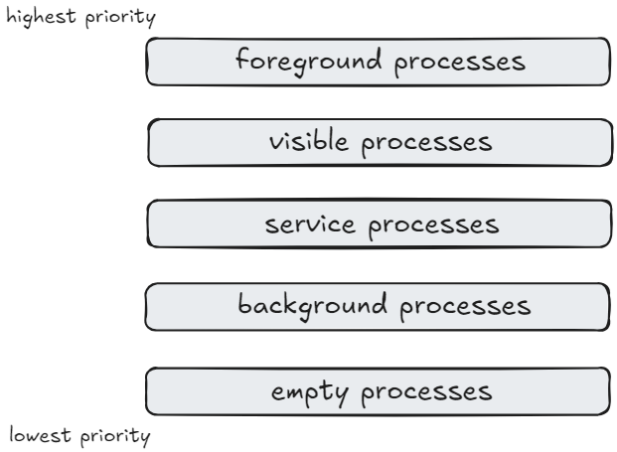

Figure 6: Android process hierarchy

As an example, the establishing of the queues can be done by following the importance hierarchy of Android processes, which is as follows:

- **Foreground process:** the current process visible on the screen, representing the application the user is currently interacting with.
- **Visible process:** a process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process).
- **Service process:** a process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music).
- **Background process:** a process that may be performing an activity but is not apparent to the user.
- **Empty process:** a process that holds no active components associated with any application.

2.7 Multilevel Feedback Queue Scheduling

This method is similar to multilevel queue scheduling, except that processes may be moved from one queue to another for a variety of reasons: if the characteristics of a process change between *CPU-intensive* and *I/O-intensive*, then it may be appropriate to switch it to another queue; *aging* can also be incorporated, so that a process that has waited for a long time can get bumped up into a higher priority queue. A multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues.
- The scheduling algorithms used for each queue.
- The methods used to determine when to **upgrade** or **demote** a process.
- The method used to determine which queue a process will enter intitially.

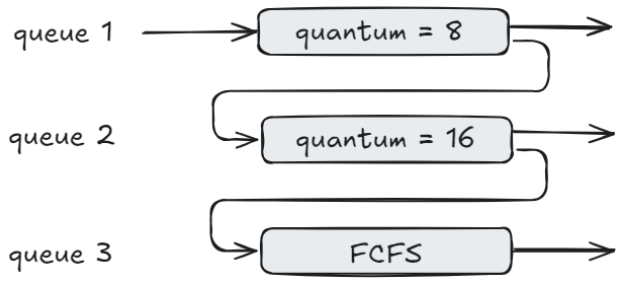

Figure 7: Multilevel feedback queue scheduling

When a new process arrives, it enters Q_1, which is served in round-robin fashion. When the process gets the CPU, it is allowed to execute for at most 8ms. If it does not finish, it is moved to Q_2, which is also served in round-robin fashion. If it does not finish within 16ms, it is preempted again and moved to Q_3.

3 Thread scheduling

On most modern operating systems, it is not processes, but **kernel-level threads** that are being scheduled by the OS. The boundary within which threads compete for CPU resources is known as the **contention scope**; it determines whether a thread is fighting for processor time against other threads in the same process or against every other thread in the system.

On systems implementing the *many-to-one* and *many-to-many* multithreading models, the thread library schedules user-level threads onto available *lightweight processes*. Thus, the competition for resources takes place locally, within a single process. This scheme is known as **process-contention scope (PCS)**.

In a **system-contention scope (SCS)** scheme, the competition for resources takes place between all threads in the system. Systems using the *one-to-one* model (every thread is mapped directly to a kernel-level thread) schedule threads using only SCS (such as Windows and Linux).

4 Multiprocessor scheduling

If multiple CPUs are available, then **load sharing** (the ability to distribute the workload across all available CPUs; the focus is to keep all resources active, not ensure an equal distribution of work) becomes a possibility. However, scheduling issues become correspondingly more complex. **Multiprocessor scheduling** is the process of deciding which thread should run on which processor at any given time in a system with more than one CPU.

On modern computing systems, the term *multiprocessor* now applies to the following system architectures: **multicore CPUs**, **multithreaded cores**, **NUMA systems** (non-uniform memory access; each CPU has its own "local" memory that can be accessed quickly), and **heterogenous multiprocessing** (instead of having identical cores, there are weaker and more powerful cores for certain tasks).

4.1 Approaches to multi-processor scheduling

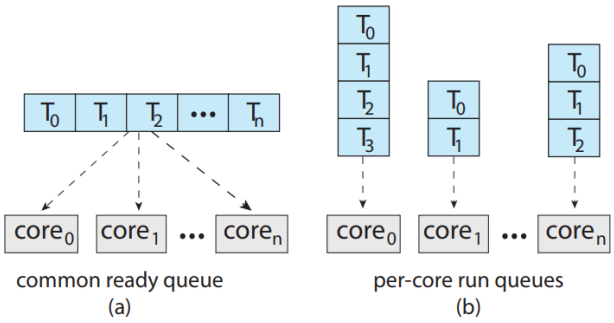

Figure 8: Organization of ready queues in a SMP system

One simple approach to CPU scheduling in a multiprocessor system, known as **asymmetric multiprocessing**, hands all the responsibilities of making scheduling decisions, I/O processing, and other system activities to a single processor (the *master server*). The other processors only execute user code.

The downfall of this approach is that the master server becomes a potential **bottleneck**; as more worker cores are added, the master core will get busier and busier assigning tasks and handling I/O; at one point, it will no longer be able to keep up.

The standard approach for supporting multiprocessor systems is **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. The scheduler for each processor examines the ready queue and selects a thread to run. This approach provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue. This may lead to **race conditions** on the shared ready queue; therefore, it must be ensured that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue. Some form of locking could be used; however, the lock would be **highly contended**, creating a *bottleneck*.

2. Each processor may have its own private queue of threads. This strategy does not suffer from the possible performance problems associated with a shared queue; thus, it is the most common approach on systems supporting SMP.

4.2 Multicore processors

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This is known as a **memory stall**, which occurs primarily because modern processors operate at much faster speeds than memory. However, a memory stall can also occur due to a *cache miss*.

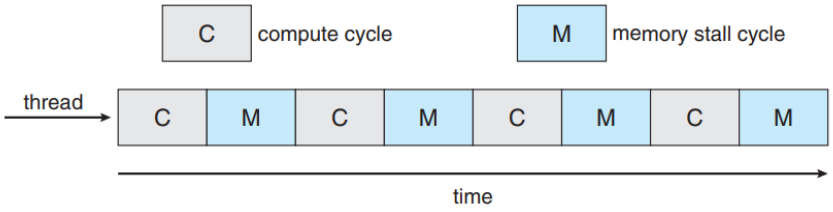

Figure 9: Memory stall

To solve this issue, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core. If one hardware thread stalls while waiting for memory, the core can switch to another thread. Each hardware thread maintains its architectural state (program counter, register set, etc.), and thus appears as a logical CPU that is available to run a software thread.

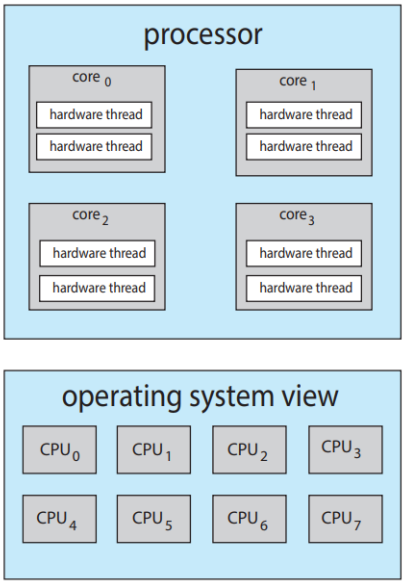

Figure 10: Chip multithreading

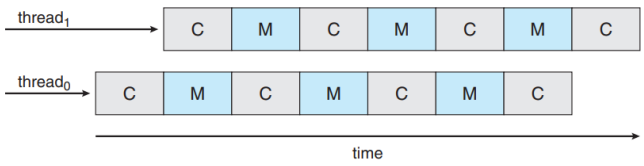

Figure 11: Multithreaded multicore system

The resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, so a processing core can only execute one hardware thread at a time. Consequently, a multithreaded multicore processor actually requires two different levels of scheduling. On one level are the scheduling decisions that must be made by the OS as it chooses which software thread to run on each hardware thread (logical CPU). For this level of scheduling, the OS may choose any scheduling algorithm described earlier.

A second level of scheduling specifies how each core decides which hardware thread to run. One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core. Another strategy is used by the *Intel Itanium*, a dual-core processor with two hardware-managed threads per core. Each hardware thread has a **dynamic urgency value** ranging from 0 to 7 (0 represents the lowest urgency, 7 represents the highest). The Itanium identifies five different events that may trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

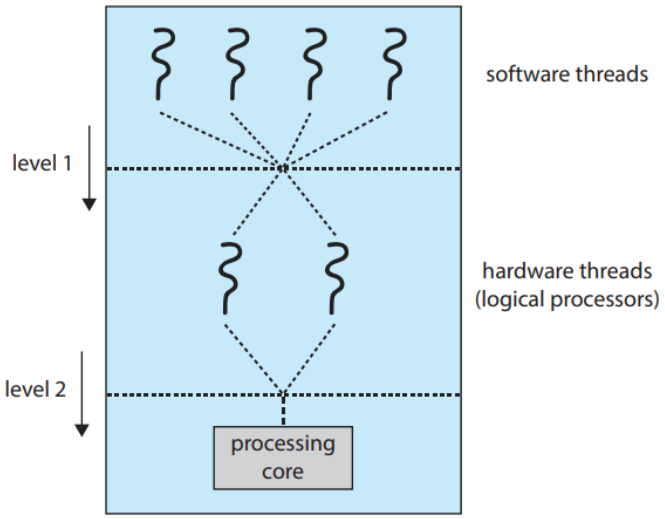

Figure 12: Two levels of scheduling

The two different levels of scheduling are not necessarily mutually exclusive. If the OS scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions. As an example, assume that a CPU has two processing cores and each core has two hardware threads. If two software threads are running on this system, they can be running either on the same core or on separate cores. If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores. If the OS is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

Intel processors use the term **hyperthreading** (also known as **simultaneous multithreading** or **SMT**) to describe assigning multiple hardware threads to a single processing core. In general, there are two ways to multithread a processing core:

1. **Coarse-grained multithreading:** a thread executes on a core until a long-latency event (such as a memory stall) occurs. Due to the delay caused by the event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.

2. **Fine-grained (interleaved) multithreading:** this strategy switches between threads typically at the boundary of an instruction cycle. As a result, the cost of switching between threads is small.

4.3 Load balancing

In a system where each processor has its own private ready queue, the workload can quickly become uneven. In SMP systems, it is important to keep the workload balanced between all processors to fully utilize the benefits of having more than one processor; otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU.

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is only necessary in systems where each processor has its own private ready queue of threads. On systems with a common ready queue, load balancing is unnecessary; once a processor becomes idle, it immediately extracts a runnable thread from the queue. There are two general approaches to achieving load balancing:

1. **Push migration:** a specific task periodically checks the load on each processor. If it finds an imbalance, it evenly distributes the load by moving (pushing) threads from overloaded to idle or less-busy processors.
2. **Pull migration:** an idle processor pulls a waiting task from a busy processor.

These approaches are not mutually exclusive. In fact, they are often implemented in parallel on load-balancing systems.

4.4 Processor affinity

When a thread continuously runs on a specific processor, the data most recently accessed by the thread populates the cache of the processor. As a result, successive memory accesses by the thread are often satisfied in cache memory (creating a **warm cache**).

If the thread migrates to another processor (e.g., due to load balancing), the contents of the cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. These are costly operations, so most systems with SMP support try to avoid migrating a thread from one processor to another; instead, they attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as **processor affinity** (a process has an affinity for the processor on which it is currently running).

The two strategies described earlier for organizing the queue of threads available for scheduling have implications for processor affinity. For a common ready queue, a thread can be selected for execution by any processor. Thus, if a thread is scheduled on a new processor, the cache of that processor must be repopulated. With private per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache, providing processor affinity.

When an OS has a policy of trying to keep a process running on the same processor, but not guaranteeing that it will do so, it is a situation known as **soft affinity**. The OS will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it can run.

The main-memory architecture of a system can also affect processor affinity. Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU. If the operating system's CPU scheduler and memory-placement algorithms are *NUMA-aware* and work together, then a thread that has been scheduled to a particular CPU can be allocated memory closest to where the CPU is, providing the thread with the fastest possible memory access.

Load balancing often counteracts the benefits of processor affinity. The advantage of keeping a thread running on the same processor is a *warm cache*; attempting to balance loads by moving a thread from one processor to another removes this benefit. Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times. In other words, there is a natural tension between load balancing and minimizing memory access times. Consequently, scheduling algorithms for modern multicore NUMA systems have become quite complex.

4.5 Heterogenous multiprocessing

Some mobile systems are now designed using cores that run the same instruction set, although vary in terms of their *clock speed* and *power management*, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as **heterogeneous multiprocessing (HMP)**. The intention is to better manage power consumption by assigning tasks to certain cores based on the specific demands of the task.

For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-peformance **big** cores are combined with energy efficient **LITTLE** cores. Big cores consume greater energy and therefore should only be used for short periods of time. Likewise, little cores use less energy and can therefore be used for longer periods.

By combining slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance but may need to run for longer periods (such as background tasks) to little cores. Similarly, applications that require more processing power but may run for shorter durations can be assigned to big cores. Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores.

5 Real-time CPU scheduling

Focusing on scheduling processes in terms of deadlines is a strategy known as **real-time CPU scheduling**. A **hard real-time system** is a system with utterly strict deadlines; failing to meet a deadline is catastrophical (e.g., medical systems). For **soft real-time systems**, deadlines are important, but occasional misses are tolerable.

5.1 Minimizing latency

When a real-time event occurs in a system (such as when a timer expires), the system must respond to and service it as quickly as possible. The **event latency** is the amount of time that elapses from when an event occurs to when it is serviced.

Different events have different latency requirements. Any response that takes longer than the requirement might result in having something go wrong with system. Two types of latencies affect the performance of real-time systems:

1. **Interrupt latency:** the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.

2. **Dispatch latency:** the amount of time required for the *scheduling dispatcher* to stop one process and start another.

The **conflict phase** is the portion of the dispatch latency where the system must resolve "conflicts" that prevent a high-priority process from running immediately. It is primarily composed of two components:

1. Preemption of any process running in the kernel.

2. Release of resources by low-priority processes, needed by a high-priority process.

5.2 Priority-based scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. Thus, the scheduler for a real-time operating system must support a **priority-based scheduling algorithm with preemption**.

Providing a preemptive priority-based scheduler only ensures *soft real-time functionality*. Hard real-time systems must further ensure that real-time tasks will be serviced with respect to their deadline requirements. Making such promises requires additional scheduling features.

Certain characteristics of the processes that are to be scheduled must be defined. Firstly, the processes are considered **periodic** (they require the CPU at constant intervals). Once a periodic process has acquired the CPU, it has a **fixed processing time** t, a **deadline** d by which it must be serviced by the CPU, and a **period** p. The relationship between these variables can be expressed as $0 \leq t \leq d \leq p$ (the work must be equal to or less than the deadline, and the deadline falls within the current period). The rate (frequency) of a periodic task is $\frac{1}{p}$. Schedulers can take advantage of these characteristics and assign priorities according to a process' deadline or rate requirements.

A process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control algorithm**, the scheduler either admits the process, guaranteeing that it will complete on time, or rejects the request as impossible if it cannot assure that the task will be serviced by its deadline.

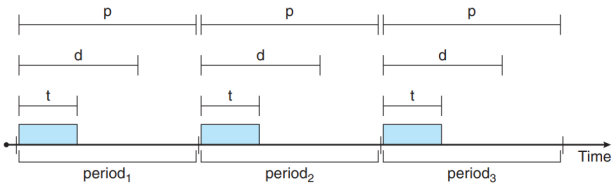

Figure 13: Periodic task